Natural Resources and Environmental Issues

Volume 8 *SwarmFest 2000* Article 4

2001

Repast: An extensible framework for agent simulation

Nick Collier
Social Science Research Computing, University of Chicago, IL

REPAST: AN EXTENSIBLE FRAMEWORK FOR AGENT SIMULATION

NICK COLLIER

Social Science Research Computing, 1155 E. 60th. St., University of Chicago, Chicago, Illinois, 60637, Email: nick@src.uchicago.edu

Abstract. Repast is a software framework for agent-based simulation created by Social Science Research Computing at the University of Chicago. It provides a library of classes for creating, running, displaying, and collecting data from an agent-based simulation. This paper provides an overview of Repast's design, features, and capabilities and describes the implementation of some of its key abstractions.

INTRODUCTION

The University of Chicago's Social Science Research Computing's Repast* is a software framework for creating agent-based simulations using the Java language. It provides a library of objects for creating, running, displaying, and collecting data from an agent-based simulation. In addition, Repast can take snapshots of running simulations and create quicktime movies of simulations. Repast borrows much from the design of the Swarm (http://www.swarm.org) simulation toolkit and can properly be termed a "Swarm-like" simulation framework. with altered assumptions. To achieve this goal, it will be necessary for Repast to provide a feast of advanced features. The version of Repast described in this paper is just the beginning of our work towards this objective.

Implicit in Repast's design is the familiar notion of a simulation as a state machine whose state is constituted by the collective states of its components. These components can be divided up into infrastructure and representation. The infrastructure is the various mechanisms native to the framework that run the simulation, display, and collect data and so forth. The representation is what the simulation modeler constructs, that is, the simulation model itself. The state of the infrastructure is then the state of the display, the state of the data collection objects and so forth. The state of the representation is the state of what is being modeled, for instance, the current values of all the agents' variables, the current value of the space or spaces in which they operate, as well as the state of any other representation objects (e.g. aggregate quasi-independent "institution" objects). The history of the simulation as a software phenomenon is the history of both these states, while the history of the simulation as a simulation is the history of the representational states. In Repast, changes to the states of the infrastructural components and the representational components occur through a single object. In short then, Repast allows a user to build a simulation as a state machine in which all the changes to the state machine occur through a single object interface. This provides clarity and extensibility both for the simulation modeler as well as the software designer seeking to extend the toolkit.

HISTORY

Repast was initially conceived of as a library of Java classes that would work together with and simplify the Swarm simulation framework. This initial conception was the result of University of Chicago researchers' concerns with the complexity of both Swarm and Objective-C and

* The name Repast is an acronym for REcursive Porous Agent Simulation Toolkit. Our long-term goal is to move beyond the representation of agents as discrete, self-contained entities in favor of a view of social actors as permeable, interleaved and mutually defining, with cascading and recombinant motives. In later releases we intend to support the modeling of belief systems, agents, organizations, and institutions as recursive social constructions. The fuller goal of the toolkit is to allow situated histories to be replayed

my respect for the maturity and elegance of the Swarm API. This notion of Repast as an extension to Swarm was soon abandoned for a variety of reasons and made partially redundant with the release of Java Swarm (a Java layer running on top of the Swarm kernel and released by the Swarm Development Group). Prior to the release of Java Swarm, I had begun some exploration into developing an independent framework completely designed in Java, but borrowing several of the key abstractions present in Swarm. Convinced of this framework's viability and usefulness to University of Chicago researchers, the initial exploration grew into Repast as it stands today. It has been associated with Swarm from its beginning, and this results in its Swarm-like appearance.

DESIGN GOALS

Repast's design goals grew out of our constituency's (University of Chicago researchers) concern with ease of use and the desire for a short learning curve, as well as my concerns about extensibility and robustness. I tried to meet these larger goals through the following design goals: abstraction of simulation infrastructure, extensibility, and "good enough" performance.

Abstraction

Repast abstracts most of the key elements of agent-based simulation and represents them as a Java class or classes. These classes cooperate to make a framework for creating agent-based simulations. Much of the design of this cooperation makes use of design patterns* and achieves some small measure of elegance and clarity due to it. The current 1.0 version of Repast provides a ready-to-use class or classes for most of the common infrastructural abstractions of an agent-based simulation (e.g., scheduling, display, data collection, and so forth) and some generic components for constructing representational elements. These generic components include such things as agent spaces, and a few generic agent types. Future work on Repast will concentrate in part on creating libraries of ready to use spaces and agents. Repast bases many of its key abstractions on those found in Swarm, including most importantly, the notion of a Schedule as the engine for driving a simulation.

Extensibility

As a design goal, extensibility grows largely from the success of the design and implementation of key abstractions. In making use of some of the Swarm abstractions Repast inherits a time-tested design that contributes to its extensibility. In implementing some of these abstractions using design patterns extensibility is again enhanced. For example, the scheduling mechanism (the Schedule object and the various action classes) is implemented according to the composite design pattern, which allows client code to treat individual actions and the compositions of those actions uniformly. This provides clarity to the scheduling mechanism and allows it to be easily extended in the future.

Extensibility is also provided by the use of Java as an implementation language. Object-oriented languages easily lend themselves to the creation of extensible frameworks through the use of inheritance and composition.

"Good Enough" Performance

"Good enough" performance refers to a level of performance that is acceptable when weighed against the other benefits of the toolkit. While performance optimizations were not part of the initial design, care was taken to minimize object creation and achieve acceptable display speed. This goal has been met and surpassed. Repast offers performance comparable to similar frameworks and should only get faster with the continual improvement of Java virtual machines.

As a result of these goals, Repast is robust, extensible, and fairly easy to use, although the modeler must still learn Java. However, choosing Java as an implementation language has its own benefits. Java eliminates the kind of memory leaks associated with C, C++, and Objective-C, a particular problem for long running simulations. Java is well documented and there are many instructional books devoted to it, and its cross

* Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass: Addison-Wesley.

platform design also makes installation and setup on a variety of platforms quite simple.

PACKAGE OVERVIEW

Repast consists of approximately 130 classes organized into six packages as well as several demonstration simulations.

Analysis

The classes in the analysis package are used to gather, record, and chart data. Using the DataRecorder class, a modeler can specify the data to be collected and write that data to a file in a tabular format.

Engine

The engine classes are responsible for setting up, manipulating, and driving a simulation. The SimModel interface is the superclass for all models written with Repast. A partial implementation of SimModel, the SimModelImp class is provided and can be used as the base class for most, if not all, models written with Repast. The controller classes (BaseController, Controller, and BatchController) are responsible for handling user interaction with a simulation either through a GUI or by automating such interaction through the use of a batch parameter file. The Schedule class and its associated action classes are used to effect any state changes to the simulation and are described in further detail below.

Games

The games package contains a few classes for implementing prisoner's dilemma type cooperation games.

Gui

The gui classes are responsible for the graphical animated visualization of the simulation as well as providing the capability to take snapshots of this display and make quicktime movies of the visualization as it evolves over time. The various *Display classes work in conjunction with the classes in the space package to display these spaces appropriately. Through a DisplaySurface, the *Painter classes handle the actual display of these spaces on the screen, and the DisplaySurface itself allows for the probing of the displayed objects. Probing, left clicking on the visualization of a simulation object, introspects that object (an agent for example) and displays its current parameters in a separate window.

Space

The space classes are essentially container classes that represent various types of spaces (two dimensional grids, tori, and so forth) accessible through the appropriate interfaces. For example, the grid spaces allow objects to be inserted and retrieved based on x and y coordinates. The space package also contains some node and link interfaces and classes that allow the modeler to implement network-based simulations. Spaces work in conjunction with the display classes in the gui package to present a visualization of the space and the objects (e.g., agents) that it contains.

Util

Util, the utilities package, contains a single class, SimUtiltities, that provides static methods for shuffling lists, displaying information dialogs and so forth.

In addition to its own classes, Repast also makes use of those in external libraries, most notably the Colt library.(http://nicewww.cern.ch/~hoschek/colt/index.htm). The Colt library provides random number generation for Repast through its implementation of the MersenneTwister (MT19937), one of the strongest pseudo-random number generators known so far. Various other random number generators and distributions are also found in the Colt library and are thus available for use with Repast.

INSIDE REPAST

This section discusses how the scheduling and display mechanisms are implemented.

Scheduling Mechanism

The scheduling mechanism is responsible for all the user-defined state changes within a Repast simulation.* The design of Repast's scheduling

* The scheduling mechanism is responsible for any user-defined state changes. However, controller and model

mechanism is based on the observable features of the Swarm scheduling mechanism. At its heart, scheduling consists of setting up method calls on objects to occur at some specified time. Repast represents these method calls separately from the objects themselves through the BasicAction class. A BasicAction consists of some variables used by the Scheduler class and an abstract public void execute() method. Any classes that sub-class a BasicAction must implement this method, and it is in this method that the actual method call or calls to be scheduled should occur.

As mentioned above, the scheduling mechanism implements the composite design pattern. The scheduling objects are composed into a tree structure that represents a part-whole hierarchy. In this context the BasicAction class represents primitives (SimAction, SimListAction, and anonymous and otherwise user-defined classes that inherit from BasicAction) and their containers (Schedule and ActionGroup). As containers, the Schedule and ActionGroup objects primarily store the primitive or leaf components, although they can also store other Schedule or ActionGroup objects. Schedule objects store BasicActions and associated information about when to execute these BasicActions. ActionGroups allow for the grouping of BasicActions into groups of conceptually similar actions and provide methods to determine the order of execution of the BasicActions within that ActionGroup.

The operation that all these objects share is the execute method mentioned above. The primitive components implement the execute method to call the actual methods on the simulation objects, while the containers implement execute to call execute on their children. In addition to this execute method, the Schedule object contains other methods to add BasicActions to itself together with associated scheduling information. BasicActions can be scheduled to execute at some specific time, at a specified interval, or at every iteration beginning at specified time. BasicActions can themselves be created through method calls to an ActionGroup or to a Schedule object, as well as through the explicit creation of sub-classes. However, while creation of BasicActions through the container objects allows a considerable dynamism, it is considerably slower than the execution of explicitly created sub-classes of BasicAction because it uses the Java reflection mechanism. A class that sub-classes a BasicAction is typically created as an inner class and this inner class is then added to one of the container classes.

The master schedule that controls the execution of all childern will be created in a model implementing the SimModel interface either on its own or by extending SimModelImp. BasicActions will then be created and added to this Schedule object together with their associated scheduling information. One of the controller classes mentioned above then begins a loop that calls, among other things, execute on this Schedule object. At this point the time or tick count is zero. The Schedule object then builds the execution queue for the current time based on the scheduling information associated with each basic action, adding its child BasicActions to the queue if appropriate. It then iterates over this queue calling execute on the BasicActions in the queue. Consequently, the primitive or leaf BasicActions will then call methods on actual simulation objects thus changing the state of the simulation. When the Schedule object finishes iterating over the execution queue, the tick count is incremented. It is this tick count against which BasicActions are scheduled for execution.

This design and implementation provides a clear and flexible scheduling mechanism. Complicated schemes of execution can be created through the composition of BasicActions, ActionGroups, and Schedules.

Display Mechanism

The display mechanism is responsible for displaying a visualization of a running simulation in real time. The mechanism primarily consists of the space classes from the space package, the displays corresponding to those spaces, the SimGraphics class, the various drawable interfaces (Drawable, Drawable2DNode, and so forth) associated with the display and spaces classes, a

classes sometimes communicate via an event-based mechanism, the results of which may effect some infrastructural state changes.

LocalPainter and a DisplaySurface, all from the gui package. As mentioned above, spaces are topological containers for simulation objects, most likely agents. For example, the space class Object2DTorus represents a two dimensional toroidal grid where each grid cell can contain an object. The display classes contain a single space and provide an interface and implementation for displaying the objects contained within that space. If the objects within a space are to be displayed, those objects must be of a certain type. The various drawable interfaces define these types, and as interfaces they can be implemented by any type of object. The displays also implement the Probeable interface, taking screen coordinates, converting those coordinates into coordinates relevant to their topology and returning a list of objects at those coordinates. The SimGraphics class is a wrapper around java.awt.Graphics2D and as such simplifies the drawing of circles, rectangles, text, colors and so forth. The LocalPainter is a container for displays, and handles the actual drawing of these displays, double buffering, and Graphics2D manipulation. The DisplaySurface handles probing and is the public interface to the drawing mechanism and the LocalPainter in particular. (A modeler will add displays to a DisplaySurface, which then adds them to the LocalPainter). Creating a Repast display is thus a matter of deciding on a space or spaces, implementing the drawable interfaces appropriate to these spaces in the objects that the spaces contain, adding these spaces to the appropriate displays, and then adding these displays to a DisplaySurface.

Given this structure, the actual drawing sequence is as follows. The scheduling mechanism calls the updateDisplay method on the DisplaySurface object. Having received this call, the DisplaySurface object tells the LocalPainter to paint itself. The LocalPainter then creates a java.awt.Graphics2D object from an off-screen BufferedImage, and wraps a SimGraphics object around this Graphics2D object. It then calls the DrawDisplay(SimGraphics g) method on each display it contains, passing this SimGraphics object as an argument. The display then either gets a list of all the objects in the space it contains from that space, or if a list of objects was added to the display, that list is used.* The display then iterates through this list, requesting some drawing information (coordinates, size, etc.) from each object in the list through the appropriate drawable interface, and prepares the passed-in SimGraphics object using this information. Each drawable object in the list is then told to draw itself, using the passed-in SimGraphics object. When the painter has finished iterating through all the displays, it draws the off-screen image to the screen, and the drawing for that tick has finished.

The structure of the display mechanism is thus one of composition where each container delegates the actual drawing responsibilities to their children. This provides flexibility and extensibility, although the tight conceptual coupling between spaces, displays, and drawable interfaces means that such extensibility requires the implementation of several classes and interfaces.

THE FUTURE

Current work on Repast concerns expanding the visualization capabilities into three dimensions and working on ease of use. To promote greater ease of use, a Repast IDE (integrated development environment) is under development, although still at the early stages. Part of this effort is the development of a library of "standard" agents and environments that can be customized by the modeler. In addition we continue to lay the foundations for the longer-term goals described above.

* In most cases, the modeler has the option of adding a list of objects to be displayed directly to the displays together with the space. The list of objects received from the space may often contain null objects if that space is sparsely populated. Consequently, it is frequently more efficient to use the passed-in list, rather than that received from the space.